

Synthesizing Finite-state Protocols from Scenarios and Requirements^{*}

Rajeev Alur¹, Milo Martin¹, Mukund Raghothaman¹, Christos Stergiou^{1,2},
Stavros Tripakis^{2,3}, and Abhishek Udupa¹

¹ University of Pennsylvania

² University of California, Berkeley

³ Aalto University

Abstract. Scenarios, or Message Sequence Charts, offer an intuitive way of describing the desired behaviors of a distributed protocol. In this paper we propose a new way of specifying finite-state protocols using scenarios: we show that it is possible to automatically derive a distributed implementation from a set of scenarios augmented with a set of safety and liveness requirements, provided the given scenarios adequately *cover* all the states of the desired implementation. We first derive incomplete state machines from the given scenarios, and then synthesis corresponds to completing the transition relation of individual processes so that the global product meets the specified requirements. This completion problem, in general, has the same complexity, PSPACE, as the verification problem, but unlike the verification problem, is NP-complete for a constant number of processes. We present two algorithms for solving the completion problem, one based on a heuristic search in the space of possible completions and one based on OBDD-based symbolic fixpoint computation. We evaluate the proposed methodology for protocol specification and the effectiveness of the synthesis algorithms using the classical alternating-bit protocol.

1 Introduction

In formal verification, a system model is checked against correctness requirements to find bugs. Sustained research in improving verification tools over the last few decades has resulted in powerful heuristics for coping with the computational intractability of problems such as Boolean satisfiability and search through the state-space of concurrent processes. The advances in these analysis tools now offer an opportunity to develop new methodologies for system design that allow a programmer to specify a system in more intuitive ways. In this paper, we focus on distributed protocols: the multitude of behaviors arising due to asynchronous concurrency makes the design of such protocols difficult, and the benefits of using model checkers to debug such protocols have been clearly demonstrated. Traditionally a distributed protocol is described using communicating finite-state

^{*} This is the working draft of a paper currently in submission. (February 10, 2014)

machines (FSMs), and the goal of this paper is to develop a methodology aimed at simplifying the task of specifying them.

A more intuitive way of specifying the desired behaviors of a protocol is by *scenarios*, where each scenario describes an expected sequence of message exchanges among participating processes. Such scenarios are used in textbooks and classrooms to explain the protocol and can be specified using the intuitive visual notation of Message Sequence Charts. In fact, the MSC notation is standardized by IEEE [1], and it is supported by some system development environments as design supplements. These observations raise the question: is it plausible to ask the designer to provide enough scenarios so that the protocol implementation can be automatically synthesized? Although one cannot expect a designer to provide scenarios that include all the possible behaviors, our key insight is that even a *representative* set of scenarios *covers* all the states of the desired implementation. From a scenario, (local) states of a process are obtained from explicit state-labels that appear as annotations as well as from the histories of events in which the process participates. If we consider all the states and the input/output transitions out of these states for a given process that appear in the given set of scenarios, we obtain a *skeleton* of the desired FSM implementation of that process. The synthesis problem now corresponds to *completing* this skeleton by adding transitions. This requires the synthesizer to infer, for instance, how to process a particular input event in a particular state even when this information is missing from the specified scenarios. The more such completions that the synthesizer can learn successfully, the lower the burden on the designer to specify details of each and every case. To rule out incorrect completions, we ask the designer to provide a model of the environment and correctness requirements. Some requirements such as absence of deadlocks can be generic to all the protocols, whereas other requirements specific to the coordination problem being solved by the protocol are given as finite-state monitors for safety and liveness properties commonly used in model checkers.

The synthesis problem then maps to the following *protocol completion* problem: given (1) a set of FSMs with incomplete transition functions, (2) a model of the environment, and (3) a set of safety/liveness requirements, find a completion of the FSMs so that the composition satisfies all the requirements. We show this problem, similar to the model checking problem, to be PSPACE-complete, but, unlike the model checking problem, to be NP-hard for just one process. We focus on two approaches to solving this problem: the first performs a search through the space of possible completions with heuristics guiding the search order and the second uses OBDD-based symbolic model checking to compute the set of correct completions by encoding the unknown targets of transitions as rigid variables.

To evaluate our methodology, we consider the Alternating Bit Protocol, a classical solution to provide reliable transmission using unreliable channels. The canonical description of the protocol [2] uses four scenarios to explain its behavior. It turns out that the first scenario corresponding to the typical behavior contains a representative of each local state of both the sender and receiver processes. Our symbolic algorithm for protocol completion is able to find the

correct implementation from just one scenario, and thus, automatically learn how to cope with message losses and message duplications. We then vary the input, both in terms of the set of scenarios and the set of correctness requirements, and study how it affects the computational requirements and the ability to learn the correct protocol for both the completion algorithms.

Related Work

Our work builds on techniques and tools for model checking [3] and also on the rich literature for formal modeling and verification of distributed protocols [4].

The problem of deriving finite-state implementations from formal requirements specified, for instance, in temporal logic, is called *reactive synthesis*, and has been studied extensively [5,6,7]. When the implementation is required to be distributed, the problem is known to be undecidable [8,9,10,11]. In *bounded synthesis*, one fixes a bound on the number of states of the implementation, and this allows algorithmic solutions to distributed synthesis [12]. Another approach uses *genetic programming*, combined with model checking, to search through protocol implementations to find a correct one, and has been shown to be effective in synthesizing protocols such as leader election [13,14].

Specifying a reactive system using example scenarios has also a long tradition. In particular, the problem of deriving an implementation that exhibits at least the behaviors specified by a given set of scenarios is well-studied (see, for instance, [15,16]). A particularly well-developed approach is *behavioral programming* [17] that builds on the work on an extension of message sequence charts, called *live sequence charts* [18], and has been shown to be effective for specifying the behavior of a single controller reacting with its environment. More recently, scenarios — in the form of “flows” — have been used in the modular verification of cache coherence protocols [19].

Our approach of using *both* the scenarios and the requirements in an integrated manner, and using scenarios to derive incomplete state machines, offers a conceptually new methodology compared to the existing work. We are inspired by recent work on program sketching [20] and on protocol specification [21]. Compared to TRANSIT [21], in this paper we limit ourselves to finite-state protocols, but consider both safety and liveness requirements, and provide a *fully automatic* synthesis procedure.

The protocol completion problem itself has conceptual similarities to problems such as *program repair* studied in the literature [22], but differs in technical details.

2 Methodology

We explain our methodology by illustrating it on an example, the well-known Alternating Bit Protocol (ABP). The ABP protocol ensures reliable message transmission over unreliable channels which can duplicate or lose messages. As input to the synthesis tool the user provides the following:

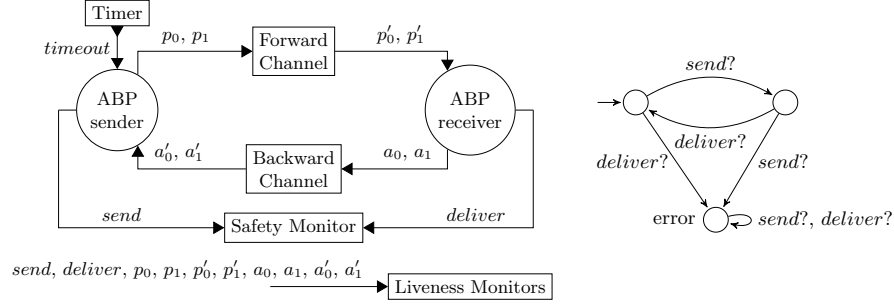


Fig. 1. ABP system architecture (left) and the safety monitor which ensures that send and deliver messages alternate (right)

- The *protocol skeleton*: this is a set of processes which are to be synthesized, and for each process, the *interface* of that process, i.e., its inputs and outputs.
- The *environment*: this is a set of processes which are known and fixed, that is, are not to be synthesized nor modified in any way by the synthesizer. The environment processes interact with the protocol processes and the product of all these processes forms a *closed* system, which can be model-checked against a formal specification.
- A *specification*: this is a set of formal requirements. These can be expressed in different ways, e.g., as temporal logic formulas, safety or liveness (e.g., Büchi) monitors, or “hardwired” properties such as absence of deadlock.
- A set of *scenarios*: these are example behaviors of the system. In our framework, a scenario is a type of *message sequence chart* (MSC).

In the case of the ABP example, the above inputs are as follows. The overall system is shown in Figure 1. The protocol skeleton consists of the two unknown processes *ABP Sender* and *ABP Receiver*. Their interfaces are shown in the figure, e.g., ABP Sender has inputs a'_0, a'_1 , and $timeout$ and outputs $send, p_0$, and p_1 . The environment processes are: *Forward Channel* (FC) (from ABP Sender to ABP Receiver, duplicating and lossy), *Backward Channel* (BC) (from ABP Receiver to ABP Sender, also duplicating and lossy), *Timer* (sends $timeout$ messages to ABP Sender), *Safety Monitor*, and a set of *Liveness Monitors*.

As specification for ABP we will use the following requirements: (1) deadlock-freedom, i.e., absence of reachable global deadlock states (in the product system); (2) safety, captured by a safety monitor such as the one depicted in Figure 1; (3) Büchi liveness monitors, that accept incorrect infinite executions in which either a send message is not followed by a deliver, a deliver is not followed by a send, or a send never appears, provided that the channels are fair; as well as (4) a property that we call *non-blockingness*, which informally requires that in every reachable global state, if a process wants to send a message to another process, then the latter must be able to receive it. Non-blockingness allows to specify that the system does not have local deadlocks, where a process is blocked from making progress while the system as a whole is not deadlocked.

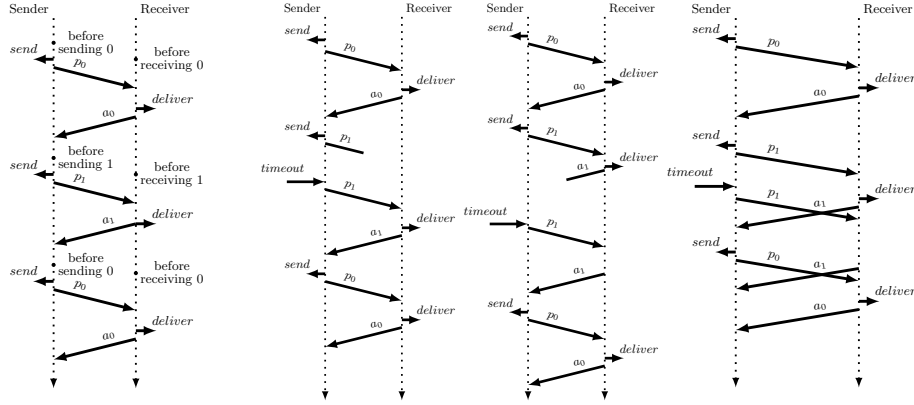


Fig. 2. Four scenarios for the alternating-bit protocol. From left to right: No loss, Lost packet, Lost ACK, Premature timeout/duplication.

We will use the four message sequence charts shown in Figure 2 to describe the behavior of the ABP protocol. They come from a textbook on computer networking [2]. The first scenario describes the behavior of the protocol when no packets or acknowledgments are lost or duplicated. The second and the third scenarios correspond to packet and acknowledgment loss respectively. Finally, the fourth scenario describes the behavior of ABP on premature timeouts and/or packet duplication.

A candidate solution to the ABP synthesis problem is a pair of processes, one for the ABP Sender and one for the ABP Receiver. Such a candidate is a valid solution if: (a) the two processes respect their I/O interface and satisfy some additional requirements such as determinism (these are defined formally in Section 3.1), (b) the overall ABP system (product of all processes) may exhibit each of the input scenarios, and (c) it satisfies the correctness requirements.

The output of the BDD-based algorithm when run with the requirements mentioned above and only the first scenario from Figure 2 is shown in Figure 4. It can be checked that these solutions are “similar/equivalent” in the sense that they satisfy the same intuitive properties that one expects from the ABP protocol. In particular, the computed solution in Figure 4 eagerly retransmits the appropriate packet when an unexpected acknowledgment is received. This behavior might incur additional traffic but satisfies all the safety and liveness properties for the ABP protocol. The computed solution for the ABP receiver is the same as the manually constructed automaton shown in Figure 3.

3 The Automata Completion Problem

We now describe how the problem, which we have set up in Section 2, can be viewed as a problem of completing the transition relations of finite IO automata.

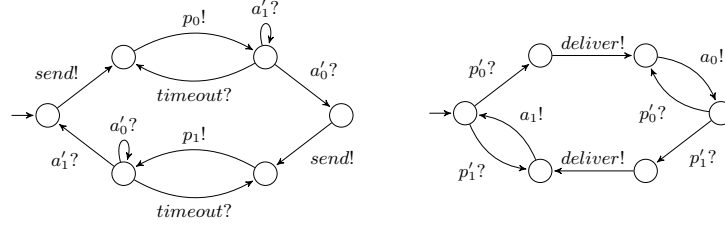


Fig. 3. ABP “manual” solution: ABP Sender (left), ABP Receiver (right).

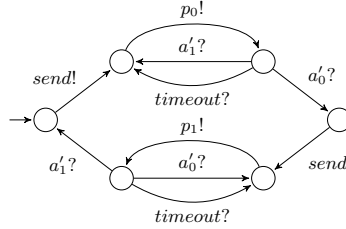


Fig. 4. Solution computed for ABP Sender by the BDD-based symbolic algorithm using only the first scenario.

3.1 Finite-state Input-Output Automata

A *finite-state input-output automaton* is a tuple $A = (Q, q_0, I, O, T)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, I is a finite (possibly empty) set of inputs, O is a finite (possibly empty) set of outputs, with $I \cap O = \emptyset$ and $T \subseteq Q \times (I \cup O) \times Q$ is a finite set of transitions⁴.

We write a transition $(q, x, q') \in T$ as $q \xrightarrow{x} q'$ when $x \in I$, and as $q \xrightarrow{x!} q'$ when $x \in O$. We write $q \rightarrow q'$ if there exists x such that $(q, x, q') \in T$. A transition labeled with $x \in I$ (respectively, $x \in O$) is called an *input transition* (respectively, an *output transition*).

A state $q \in Q$ is called a *deadlock* if it has no outgoing transitions. q is called an *input state* if it has at least one outgoing transition, and all outgoing transitions from q are input transitions. q is called an *output state* if it has a single outgoing transition, which is an output transition.

Automaton A is called *deterministic* if for every state $q \in Q$, if there are multiple outgoing transitions from q , then all these transitions must be labeled with distinct inputs. Determinism implies that every state $q \in Q$ is a deadlock, an input state, or an output state.

Automaton A is called *closed* if $I = \emptyset$.

A *safety monitor* is an automaton equipped with a set of *error states* Q_e , $A = (Q, q_0, I, O, T, Q_e)$. A *liveness monitor* is an automaton equipped with a set of *accepting states* Q_a , $A = (Q, q_0, I, O, T, Q_a)$. A monitor could be both safety and liveness, in which case it is a tuple $A = (Q, q_0, I, O, T, Q_e, Q_a)$.

⁴ The framework and synthesis algorithms can easily be extended to handle internal transitions as well, but we suppress this detail for simplicity of presentation.

A *run* of an automaton A is a finite or infinite sequence of transitions starting from the initial state: $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \dots$. A state q is called *reachable* if there exists a finite run reaching that state: $q_0 \rightarrow q_1 \rightarrow \dots \rightarrow q$. A safety automaton is called *safe* if it has no reachable error states. An infinite run of a liveness automaton is called *accepting* if it visits accepting states infinitely often. A liveness automaton is called *empty* if it has no infinite accepting runs.

3.2 Composition

We define an asynchronous (interleaving-based) parallel composition operator with rendezvous synchronization. Given two automata $A_1 = (Q_1, q_{0,1}, I_1, O_1, T_1)$ and $A_2 = (Q_2, q_{0,2}, I_2, O_2, T_2)$, the composition of A_1 and A_2 , denoted $A_1 \parallel A_2$, is defined, provided $O_1 \cap O_2 = \emptyset$, as the automaton

$$A_1 \parallel A_2 \hat{=} (Q_1 \times Q_2, (q_{0,1}, q_{0,2}), (I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2, T)$$

where $((q_1, q_2), x, (q'_1, q'_2)) \in T$ iff one of the following holds:

- $x \in O_1$ and $q_1 \xrightarrow{x!} q'_1$ and either $x \in I_2$ and $q_2 \xrightarrow{x?} q'_2$ or $x \notin I_2$ and $q'_2 = q_2$.
- $x \in O_2$ and $q_2 \xrightarrow{x!} q'_2$ and either $x \in I_1$ and $q_1 \xrightarrow{x?} q'_1$ or $x \notin I_1$ and $q'_1 = q_1$.
- $x \in (I_1 \cup I_2) \setminus (O_1 \cup O_2)$ and at least one of the following holds: (1) $x \in I_1 \setminus I_2$ and $q_1 \xrightarrow{x?} q'_1$ and $q'_2 = q_2$, (2) $x \in I_2 \setminus I_1$ and $q_2 \xrightarrow{x?} q'_2$ and $q'_1 = q_1$, (3) $x \in I_1 \cap I_2$ and $q_1 \xrightarrow{x?} q'_1$ and $q_2 \xrightarrow{x?} q'_2$.

During composition, the product automaton $A_1 \parallel A_2$ “inherits” the safety and liveness properties of each of its components. Specifically, a product state (q_1, q_2) is an error state if either q_1 or q_2 are error states. A product state (q_1, q_2) is an accepting state if either q_1 or q_2 is an accepting state.

Note that \parallel is commutative and associative. So we can write $A_1 \parallel A_2 \parallel \dots \parallel A_n$ without parentheses, for a set of n automata.

We call a product $A_1 \parallel \dots \parallel A_n$ *strongly non-blocking* if in each reachable global state (s_1, \dots, s_n) , if some automaton A_i is willing to send a message x , and all automata A_j which accept x are in non-output states, then these automata should be able to synchronize on the transition x in that global state. On the other hand, we call a product *weakly non-blocking* if in each reachable global state (s_1, \dots, s_n) , if some automaton A_i is willing to send a message x , then it is possible for all automata which accept x to eventually synchronize on the transition x . Note that strong non-blockingness is a safety property and can be useful when the model-checker cannot verify liveness properties.

3.3 From Scenarios to Incomplete Automata

The first step in our synthesis method is to automatically generate from the set of input scenarios an *incomplete automaton* for each protocol process. The second step is then to complete these incomplete automata to derive a complete protocol. In the sections that follow, we formalize and study the automata completion

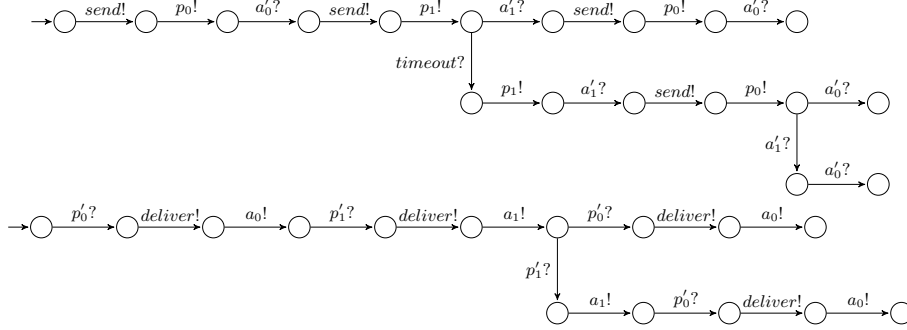


Fig. 5. Incomplete protocol automata for ABP: Sender (top), Receiver (bottom).

problem. In this section, we illustrate the first step of going from scenarios to incomplete automata, by means of the ABP example.

The idea for transforming scenarios into incomplete automata is simple. First, for every “swim lane” in the message sequence chart corresponding to a given scenario, we identify the corresponding automaton in the overall system. For example, in each scenario shown in Figure 2, the left-most lane corresponds to ABP Sender and the right-most lane to ABP Receiver. These scenarios omit the environment processes for simplicity. In particular channel processes are omitted, however, we will use a primed version of a message when referencing it on the process that receives it.

Second, for every protocol process P , we generate an incomplete automaton A_P as follows. For every message *history* ρ (i.e., finite sequence of messages received or sent by the process) specified in some scenario in the lane for P , we identify a state s_ρ in A_P . If $\rho' = \rho \cdot x$ is an extension of history ρ by one message x , then there is a transition $s_\rho \xrightarrow{x} s_{\rho'}$ in A_P . Applying this procedure to the four scenarios of Figure 2, we obtain the two incomplete automata shown in Figure 5.

Third, scenarios are annotated with labels. As shown in the first scenario of Figure 2, labels appear between messages on swim lanes. These are used to merge the states that correspond to message histories that are followed by the same label. Merging occurs for states of a single scenario as well as across multiple ones if the same label is used in different scenarios. If consistent labels are given to the initial and final positions in all swim lanes of the scenarios the resulting incomplete automata can be made cyclic. Furthermore, labels are essential for specifying recurring behaviors in scenarios and the structure of the incomplete automaton depends on the number and positions of labels used.

Finally, it is often the case that different behaviors of a system are equivalent up to simple replacement of messages. For example, all the ABP scenarios express valid behaviors if p_0 and a_0 messages are consistently replaced with p_1 and a_1 messages respectively and vice versa. Thus, our framework allows for scenarios to be characterized as “symmetric”.

We annotate the swim lanes of the ABP Sender scenarios of Figure 2 with “before sending 0” and “before sending 1” labels, and the swim lanes of the ABP

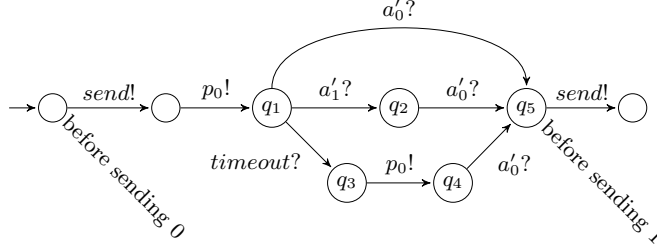


Fig. 6. Incomplete protocol automaton for ABP Sender after adding symmetric scenarios and merging labeled states. (Only the first half of the automaton is shown, the rest is the symmetric case for packet 1.)

Receiver with “before receiving 0” and “before receiving 1” labels. We also add the symmetric scenarios by switching 0 messages with 1 messages. The resulting incomplete automaton for ABP Sender is shown in Figure 6.

3.4 Automata Completion

Having transformed the input scenarios into incomplete automata, the next step is to *complete* those automata by adding the appropriate transitions, so as to synthesize a complete and correct protocol. In this section we formalize this completion problem. We define two versions of the problem: a general version (Problem 2) and a special version with only a single incomplete automaton (Problem 1). We will use Problem 1 in Section 4.1 to show that even in the simplest case automaton completion is combinatorially hard.

Consider an automaton $A = (Q, q_0, I, O, T)$. Given a set of transitions $T' \subseteq Q \times (I \cup O) \times Q$, the *completion of A with T'* is the new automaton $A' = (Q, q_0, I, O, T \cup T')$.

Problem 1. Given automaton E (the *environment*) and deterministic automaton P (the *process*) such that $E \parallel P$ is defined, find a set of transitions T such that, if P' is the completion of P with T , then P' is deterministic and $E \parallel P'$ has no reachable deadlock states.

Note that if $E \parallel P$ is defined then $E \parallel P'$ is also defined, because, by definition, completion does not modify the interface (sets of inputs and outputs) of an automaton.

Problem 2. Given a set of *environment automata* E_1, \dots, E_m and set of deterministic *process automata* P_1, \dots, P_n such that $E_1 \parallel \dots \parallel E_m \parallel P_1 \parallel \dots \parallel P_n$ is defined, find sets of transitions T_1, \dots, T_n such that, if P'_i is the completion of P_i with T_i , for $i = 1, \dots, n$, then

- P'_i is deterministic, for $i = 1, \dots, n$,
- if the product automaton $\Pi := E_1 \parallel \dots \parallel E_m \parallel P'_1 \parallel \dots \parallel P'_n$ is a safety automaton then it is safe,

- if Π is a liveness automaton then it is empty,
- Π has no reachable deadlock states,
- and, optionally, Π is weakly (or strongly) non-blocking.

Some of the environment processes E_i can be be safety or liveness monitors. The last requirement means that Problem 2 comes in three versions, one where strong non-blockingness is required, one where the weak version is required, and a third where none is. These are options provided by the user.

4 Solving Automata Completion

In this section, we consider procedures to solve the automata completion problem. First, we show that Problems 1 and 2 are NP-complete and PSPACE-complete respectively. Then, we present an explicit search algorithm that eagerly prunes parts of the search space and a heuristic which ranks candidate completions. Finally, we describe an algorithm which reduces automata completion to a model-checking problem for a symbolic model-checker.

4.1 Complexity

It can be shown that Problem 2 is PSPACE-complete. Note that this is not surprising, as the verification problem itself is PSPACE-complete, for safety properties of distributed protocols. However, in the special case of two processes, while verification can be performed in polynomial time, a reduction from 3-SAT shows that the corresponding completion Problem 1 is NP-complete. The proofs can be found in the appendix.

Theorem 1. *Problem 2 is PSPACE-complete and Problem 1 is NP-complete.*

4.2 Explicit search

This algorithm for solving the automata completion problem (Problem 2) is based on an explicit search over the space of possible completions, guided by various heuristics. More specifically, the algorithm explores a *search tree* in which every node is a set of added transitions T (we include in T the transitions added in all incomplete protocol automata). The children of each node T are those nodes $T' \supset T$ which contain exactly one more transition than T . The root of the tree is the empty set of transitions, which corresponds to the original input (i.e., the incomplete automata generated from the scenarios).

For every newly visited node T (including the root) a model-checking problem is solved: we form the product of all environment processes, protocol processes (with the added transitions T), and monitors, and we check the absence of deadlocks, safety, and liveness violations (and optionally also non-blockingness). The following cases are possible:

1. No violations are found. In this case, T is a correct solution, and the search terminates.

2. A safety or liveness violation is found. This means that this candidate solution T is incorrect. Moreover, any other candidate T' obtained by adding extra transitions to T , i.e., $T' \supseteq T$, will also be incorrect, by exhibiting the same violations. This is because adding extra local transitions can only add, but not remove, global transitions. This in turn implies that any reachable error state with T will also be a reachable error state with T' , so any safety violation with T will also be a safety violation with T' . Similarly, any reachable accepting cycle with T will also be a reachable accepting cycle with T' , so liveness violations cannot be removed either. In conclusion, in this case, the entire sub-tree under T can be pruned from the search.
3. No safety nor liveness violation is found, but a deadlock or blocking state is found. In this case, T is incorrect, but could potentially be made correct by adding more transitions. The search continues exploring the children of T .

The search algorithm saves every visited node T . The same node might be visited via different paths. For example, adding first t_1 , then t_2 , leads to the same node as adding first t_2 , then t_1 . To reduce the search space, the search stops (and backtracks) when it finds a node that has already been visited. This is clearly sound and complete.

The search continues only in Case 3. In this case, we use heuristics to determine in which order should the children of T be explored. The heuristic we used prioritizes the children of T according to how similar they are to existing transitions in the protocol automaton. We deem two transitions as being similar if their message and destination are the same and if their starting states already agree on some other transition. In other words, if two states handle a message by transitioning to the same state, or indistinguishable states, the heuristic extrapolates that the states should also handle other messages in the same manner. For example, in Figure 6, states q_1 and q_4 both handle message a'_0 by transitioning to state q_5 . Hence, the heuristic prioritizes the candidate transition from q_4 to q_3 on *timeout*, over say q_4 to q_2 , since q_1 also handles *timeout* by transitioning to q_3 . Note that this transition correctly generalizes the behavior on a single timeout described in the scenarios to multiple timeouts.

4.3 BDD-based Symbolic Computation

This technique reduces the automata completion problem to an instance of a model-checking query, which is then solved by using BDD-based symbolic model-checking techniques. Consider a set of environment automata $\{E_1, E_2, \dots, E_m\}$ and a set of (possibly incomplete) deterministic process automata $\{P_1, P_2, \dots, P_n\}$, with each $P_i = (Q_i, q_{0,i}, I_i, O_i, T_i)$, as described in Section 3. For each state $q_i \in Q_i$ and for each event $x \in I_i \cup O_i$ such that $q_i \xrightarrow{x} q'_i \notin T$ for any $q'_i \in Q_i$, we introduce a variable $t_{q_i, x}$ whose value ranges over the set $Q_i \cup \{\perp\}$. Intuitively, these variables encode all possible ways to complete the transition relation T_i , including the possibility that no transition exists. Each of the transition relations T_i is now parametrized by the newly introduced variables $t_{q_i, x}$. Let P'_i be the

automata obtained by replacing the transition relation of P_i with the parametrized transition relation whose construction we have just described.

We denote the composition $E_1 \| E_2 \| \dots \| E_m \| P'_1 \| P'_2 \| \dots \| P'_n$ by \mathcal{P} and its transition relation by \mathcal{T} . Note that \mathcal{T} is also parametrized by the newly introduced $t_{q_i, e}$ variables. Also, the original composition $E_1 \| E_2 \| \dots \| E_m \| P_1 \| P_2 \dots \| P_n$ has only one initial state, by definition, and thus, \mathcal{P} has only one (parametrized) initial state as well.

Suppose we are given a safety monitor with a set of error states. We can symbolically represent the states where the monitor automaton is in an error state by a propositional formula φ . Now, the parameter values such that states satisfying $\neg\varphi$ are not reachable in the composition \mathcal{P} can be obtained by model-checking \mathcal{P} with the CTL property, $\text{EF}\neg\varphi$. If this property is found to be true, then an erroneous state in the monitor is reachable for every valuation of the parameters. If the property is found to be false, then there must exist a valuation for the parameters which prevents a state satisfying $\neg\varphi$ from being reachable. These parameter values represent a completion that satisfies the property that no state satisfying $\neg\varphi$ is reachable.

Determinism, Deadlock Freedom, Non-blockingness and Liveness. We encode constraints for determinism by restricting the set of initial values that the parameters can take.

A deadlock state is characterized by the formula $\text{DL} = \bigwedge_{o \in \bigcup_i O_i} o_i(\neg\text{sync}(o))$, where $\text{sync}(o)$ is a formula which expresses that the (sole) sender of the event o is in a state where it can send o , and all the receivers of event o are in states where they can receive an o . Thus the formula $\text{EF}(\text{DL})$ represents the set of states which eventually deadlock. A valid completion would render these states unreachable.

Suppose we are given a liveness automaton with a set Q_a of accepting states. A completion which is live needs to have no runs that visit states in Q_a infinitely often. If all states in Q_a are represented symbolically by the propositional formula Q_a^s , then the set of states from which it is possible to visit states in Q_a infinitely often can be characterized by the CTL formula $\psi = \text{AGEF}Q_a^s$. Again, we desire that these states be unreachable in a valid completion.

The non-blockingness requirement can be expressed as a safety requirement **NonBlock**, and hence can be handled in a similar manner. Finally, we use a symbolic model-checker (NuSMV [23]) to check if \mathcal{P} satisfies $\text{EF}(\neg\varphi \vee \text{DL} \vee \neg\text{NonBlock} \vee \psi)$. The valuation of parameters for which \mathcal{P} does not satisfy this property represents the completion which has the required safety, liveness, and deadlock-freedom properties.

5 Evaluation

We investigate (1) how effective scenarios are in reducing the empirical complexity of the automata completion problem, (2) the amount of generalization that the proposed algorithms are able to perform, and (3) how adding scenarios reduces the number of formal specifications required for successful completion.

Synthesis with no scenarios. To validate our hypothesis that scenarios make the synthesis problem easier, we attempted to synthesize the ABP protocol with no transitions specified, but with bounds on the number of states of the processes. These bounds were set to be equal to the corresponding number of states in the manually constructed version of the ABP protocol. We required that the protocol satisfy all the properties discussed in Section 2.

The BDD-based symbolic algorithm ran out of memory⁵ and failed to synthesize a protocol. Recall that the heuristic algorithm performs generalization by using a similarity metric between states. When the starting point is an empty transition relation, there is no similarity between states, and the heuristic fails to differentiate between candidate transitions. The resulting search procedure runs out of time.

Varying the number of scenarios. When all four of the scenarios in Figure 2 are used, both the explicit search algorithm and the BDD-based search are able to find a correct completion for the protocol. Moreover, both the algorithms find a correct completion, when applied to just *one* scenario. A quantitative summary of our experiments can be found in Table 1. We applied our algorithms on the incomplete automata constructed from the first scenario (row 1), the second scenario (row 2), and all four scenarios (row 3). For each case, we report the number of states of the incomplete automata, the number of transitions in the completions found by the algorithms, and their computational requirements.

In the case of all four scenarios, the ranking heuristic described in Section 4.2 chooses a candidate transition that is part of the correct completion at every step. The search does not backtrack, does not prune any nodes, and takes less than a minute to complete. When only the first scenario is used, the ranking heuristic is less effective in the same way as when no scenarios were used. However, the additional structure imposed by the scenarios significantly constrains the search space and the explicit search with pruning successfully returns a completion. The incomplete automata that correspond to scenario 2 include intermediate states that the heuristic can use to successfully rank candidate edges. The runtime of the explicit search algorithm varies with the order in which candidate transitions are explored. We report the 75th percentile over several randomly chosen orders.

In contrast, the BDD-based search performs better when a single scenario is used rather than all four. The intuition behind this is that the number of BDD variables is smaller when there are fewer intermediate states in the incomplete automata. As a result, the search finishes in less than 15 seconds when only the first scenario of Figure 2 is used, less than 15 minutes when only the second scenario is used, and less than 35 minutes when all four are used. Synthesis takes longer with the second scenario because the incomplete sender automaton, as mentioned earlier, has more intermediate states than the first scenario.

Generalization and inference of unspecified behaviors. With just one scenario specified, the algorithms successfully perform the generalization required

⁵ We used a memory limit of 16GB and a time limit of one hour.

Table 1. Quantitative summary of experiments.

	Number of states in incomplete automaton		Number of transitions to be completed	Computational requirements		
	Sender	Receiver		Heuristic time	BDD time	BDD memory
Scenario 1	6	6	6	4 min.	15 sec.	100MB
Scenario 2	10	6	8	30 sec.	15 min.	1GB
All scenarios	12	8	8	45 sec.	35 min.	3.8GB

to obtain a correct completion. We believe that the generalization performed is non-obvious: the correct protocol behaviors on packet loss, loss of acknowledgments and message duplication are inferred, even though the scenario does not describe what needs to happen in these situations. As can be seen in Figure 7, the incomplete automata constructed from the scenario only describe the protocol behavior over lossless channels. The algorithms are guided solely by the liveness and safety specifications to infer the correct behavior. In contrast, when all four scenarios are specified, the scenarios already contain information about the behavior of the protocol when a single packet loss or a single message duplication occurs. The algorithm thus needs to only generalize this behavior to handle an arbitrary number of packet losses and message duplications.

Varying the Correctness Requirements. We observed that when fewer scenarios were used, we needed to specify more properties — some of which were non-obvious — so that the algorithms could converge to a correct completion. For instance, when only one scenario was specified, we needed to include the liveness property that every deliver message was eventually followed by a send message. Owing to the structure of the incomplete automata, this property was not necessary to obtain a correct completion when all four scenarios were specified. Another property which was necessary to reject trivial completions when no scenarios were specified was that there has to be at least one send message in every run. Therefore, in some cases, using scenarios can compensate for the lack of detailed formal specifications.

Discussion of Experimental Results. The experimental results clearly demonstrate that specifying the behavior of the protocol using scenarios is essential for the algorithms we have evaluated to be able to construct a correct completion. In particular, even providing just one scenario allows the algorithms to converge on a correct completion within a reasonable amount of time. In contrast, none of the approaches we have evaluated were successful in synthesizing the required protocol when no scenarios were provided. An interesting trend that we observed was that the heuristic algorithm and the BDD-based symbolic algorithm seem to complement each other. Specifically, the BDD-based symbolic algorithm was

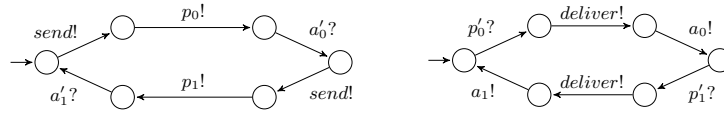


Fig. 7. Incomplete automata constructed from the first scenario of Figure 2.

effective when the number of scenarios — and therefore the number of states in the incomplete automata — was small. On the other hand, the heuristic search does better when more information is provided in the form of additional scenarios — which in turn causes a larger number of states in the incomplete automata to be similar. This is because the BDD-based symbolic algorithm essentially constructs BDDs for *all* possible completions and picks one that satisfies the properties. In general, the number of possible completions increases with the number of states in the automata, which explains why the BDD-based algorithm performs well when the number of states in the automaton is small. The heuristic algorithm exploits similarities between states to converge on a correct completion. When the algorithm is provided with the incomplete automaton from the first scenario — which has a minimal number of states — it is unable to find states which are similar and thus degenerates to an exhaustive search over all completions. Finally, we also observe that additional scenarios can compensate for unspecified correctness properties. This frees the protocol designer from having to specify the requirements completely formally.

6 Conclusions

The main contribution of this paper is a new methodology, supported by automatic synthesis techniques, for specifying finite-state distributed protocols using a mix of representative behaviors and correctness requirements. The synthesizer derives a skeleton of the state machine for each process using the states that appear in the scenarios and then finds a completion that satisfies the requirements. The promise of the proposed method is demonstrated by the ability of the synthesizer to learn the correct ABP protocol from just a single scenario corresponding to the typical case. Future research should focus on the scalability of the algorithms for protocol completion. One idea is to heuristically limit the choices for targets of transitions before applying the BDD-based symbolic computation, and another approach is to reduce the number of states by representing the implementation as an *extended* FSM with variables.

References

1. ITU Telecommunication Standardization Sector: ITU-R recommendation Z.120, Message Sequence Charts (MSC '96). (May 1996)
2. Kurose, J.F., Ross, K.W.: Computer Networking: A Top-Down Approach. 5th edn. Addison-Wesley Publishing Company, USA (2009)
3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2000)

4. Lynch, N.: Distributed algorithms. Morgan Kaufmann (1996)
5. Ramadge, P., Wonham, W.: The control of discrete event systems. *IEEE Transactions on Control Theory* **77** (1989) 81–98
6. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*. (1989)
7. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa’ar, Y.: Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.* **78**(3) (2012) 911–938
8. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: *31st Annual Symposium on Foundations of Computer Science*. (1990) 746–757
9. Tripakis, S.: Undecidable Problems of Decentralized Observation and Control on Regular Languages. *Information Processing Letters* **90**(1) (April 2004) 21–28
10. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: *IEEE Symposium on Logic in Computer Science*. (2005) 321–330
11. Lamouchi, H., Thistle, J.: Effective control synthesis for DES under partial observations. In: *39th IEEE Conference on Decision and Control*. (2000) 22–28
12. Finkbeiner, B., Schewe, S.: Bounded synthesis. *Software Tools for Technology Transfer* **15**(5-6) (2013) 519–539
13. Katz, G., Peled, D.: Model checking-based genetic programming with an application to mutual exclusion. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference. LNCS 4963* (2008) 141–156
14. Katz, G., Peled, D.: Synthesizing solutions to the leader election problem using model checking and genetic programming. In: *Haifa Verification Conference*. (2009) 117–132
15. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. *IEEE Transactions on Software Engineering* **29**(7) (2003) 623–633
16. Uchitel, S., Kramer, J., Magee, J.: Synthesis of behavioral models from scenarios. *IEEE Trans. Softw. Eng.* **29**(2) (February 2003) 99–115
17. Harel, D., Marron, A., Weiss, G.: Behavioral programming. *Commun. ACM* **55**(7) (2012) 90–100
18. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. *Formal Methods in System Design* **19**(1) (2001) 45–80
19. O’Leary, J., Talupur, M., Tuttle, M.R.: Protocol verification using flows: An industrial experience. In: *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*. (Nov 2009) 172–179
20. Solar-Lezama, A., Rabbah, R., Bodik, R., Ebcioğlu, K.: Programming by sketching for bit-streaming programs. In: *Proc. 2005 ACM Conference on Programming Language Design and Implementation*. (2005) 281–294
21. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: TRANSIT: specifying protocols with concolic snippets. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. (2013) 287–296
22. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: *Computer Aided Verification, 17th International Conference. LNCS 3576* (2005) 226–238
23. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: *Computer-Aided Verification, 14th International Conference (CAV)*. LNCS 2404, Springer (2002) 359–364

Appendix: Complexity of Automata Completion

Theorem 2. *Problem 1 is in NP.*

Proof. Problem 1 is in NP since we can guess a completion, and then check whether the requirements of the problem are satisfied. Checking whether the resulting process automaton is deterministic can be done in polynomial time. Checking whether the product automaton is deadlock-free can be done in polynomial time also, because there are only two automata in the product.

Theorem 3. *Problem 1 is NP-complete.*

Proof. We will show that 3SAT is reducible to Problem 1. Together with Theorem 2 this shows that Problem 1 is NP-complete.

Let $U = \{u_1, u_2, \dots, u_n\}$ be a set of variables and $C = \{c_1, c_2, \dots, c_m\}$ be a set of clauses, such that $|c_i| = 3$ for $1 \leq i \leq m$, making up an arbitrary instance of 3SAT.

We write z_j^1, z_j^2, z_j^3 for the literals of c_j . We write v_j^i for the variable of literal z_j^i . Note that z_j^i can be equal to v_j^i or its negation $\overline{v_j^i}$.

We construct a process automaton P and an environment E with $E \parallel P$ defined, such that the following are equivalent:

- there is a set of transitions T such that if P' is the completion of P with T , then P' is deterministic and $E \parallel P'$ has no reachable deadlock states
- there is a truth assignment for U that satisfies all clauses in C .

The process automaton has an initial state q_0 , and a pair of states q_i^t and q_i^f for each variable u_i . At each step, the environment challenges the process to instantiate some variable u_i by transmitting the message x_i^D . On receipt of this message, the completed process has to transition to one of q_i^t and q_i^f . It responds with either x_i^t or x_i^f indicating the assignment made to u_i . The environment performs this challenge for each literal in each clause, and enters a deadlock state if any clause is left unsatisfied. It follows that a completion exists iff C is satisfiable.

P is an automaton $(Q^P, q_0^P, I^P, O^P, T^P)$ such that:

- $Q^P = \{q_0^P\} \cup \{q_i^t, q_i^f \mid i \in [1, n]\}$
- $I^P = \{x_s\} \cup \{x_i^D \mid i \in [1, n]\}$
- $O^P = \{x_i^t, x_i^f \mid i \in [1, n]\}$
- $T^P = \{(q_i^t, x_i^t, q_0^P), (q_i^f, x_i^f, q_0^P) \mid i \in [1, n]\} \cup \{(q_0^P, x_s, q_0^P)\}$

E is an automaton $(Q^E, q_0^E, I^E, O^E, T^E)$ such that:

- $Q^E = \{deadlock, success\} \cup \{q_{i,j}^D, q_{i,j}^V \mid i \in \{1, 2, 3\} \text{ and } j \in [1, m]\}$
- $q_0^E = q_{1,1}^D$
- $I^E = \{x_i^t, x_i^f \mid i \in [1, n]\}$
- $O^E = \{x_s\} \cup \{x_i^D \mid i \in [1, n]\}$

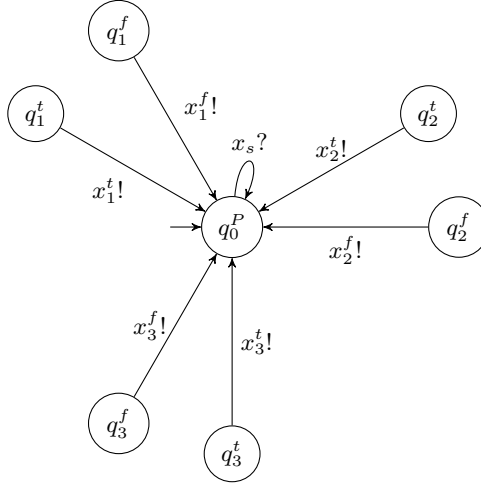


Fig. 8. Process automaton before completion. Choosing an assignment to a boolean variable u_i of the SAT problem means choosing whether to add an input transition labeled $x_i^D?$ from q_0^P to q_i^t (in which case u_i is assigned to *true*) or to q_i^f (in which case u_i is assigned to *false*).

- T^E is the smallest set such that the following hold:
 - for $i \in \{1, 2\}$ and $j \in [1, m - 1]$, $(q_{i,j}^V, x_k^f, q_{i+1,j}^D), (q_{i,j}^V, x_k^t, q_{1,j+1}^V) \in T_E$ if $z_j^i = u_k$,
 - for $i \in \{1, 2\}$, $(q_{i,m}^V, x_k^f, q_{i+1,m}^D), (q_{i,j}^V, x_k^t, success) \in T_E$ if $z_m^i = u_k$,
 - $(q_{3,m}^V, x_k^t, success)$ and $(q_{3,m}^V, x_k^f, deadlock)$ if $z_m^i = u_k$,
 - for $i \in \{1, 2\}$ and $j \in [1, m - 1]$, $(q_{i,j}^V, x_k^t, q_{i+1,j}^D), (q_{i,j}^V, x_k^f, q_{1,j+1}^V) \in T_E$ if $z_j^i = \overline{u_k}$,
 - for $i \in \{1, 2\}$, $(q_{i,m}^V, x_k^t, q_{i+1,m}^D), (q_{i,j}^V, x_k^f, success) \in T_E$ if $z_m^i = \overline{u_k}$,
 - $(q_{3,m}^V, x_k^f, success)$ and $(q_{3,m}^V, x_k^t, deadlock)$ if $z_m^i = \overline{u_k}$, and
 - $(success, x_s, success) \in T_E$.

Note that $|Q^P| = |T^P| = 2 \cdot n + 1$, $|Q^E| = 6 \cdot m + 2$, and $|T^E| = 9 \cdot m + 1$, and P and E can be constructed in polynomial time.

For $U = \{u_1, u_2, u_3\}$, $C = \{c_1, c_2\}$, $c_1 = \{u_1, \overline{u_2}, u_3\}$, and $c_2 = \{\overline{u_1}, \overline{u_2}, u_3\}$, the process automaton P and the environment automaton are shown in Figure 8 and in Figure 9 respectively.

We now prove the correctness of the reduction:

3SAT \implies Problem 1: Assume that there is a truth assignment t for U that satisfies all clauses in C .

Let T be the set of transitions such that $(q_0^P, x_i^D, q_i^t) \in T$ iff $t(u_i) = \text{true}$ and $(q_0^P, x_i^D, q_i^f) \in T$ iff $t(u_i) = \text{false}$.

Let P' be the completion of P with T . It is easy to see that every transition leaving state q_0^P has a distinct label, since we add exactly one transition for

Clause $c_1 = \{u_1, \overline{u_2}, u_3\}$ Clause $c_2 = \{\overline{u_1}, \overline{u_2}, u_3\}$

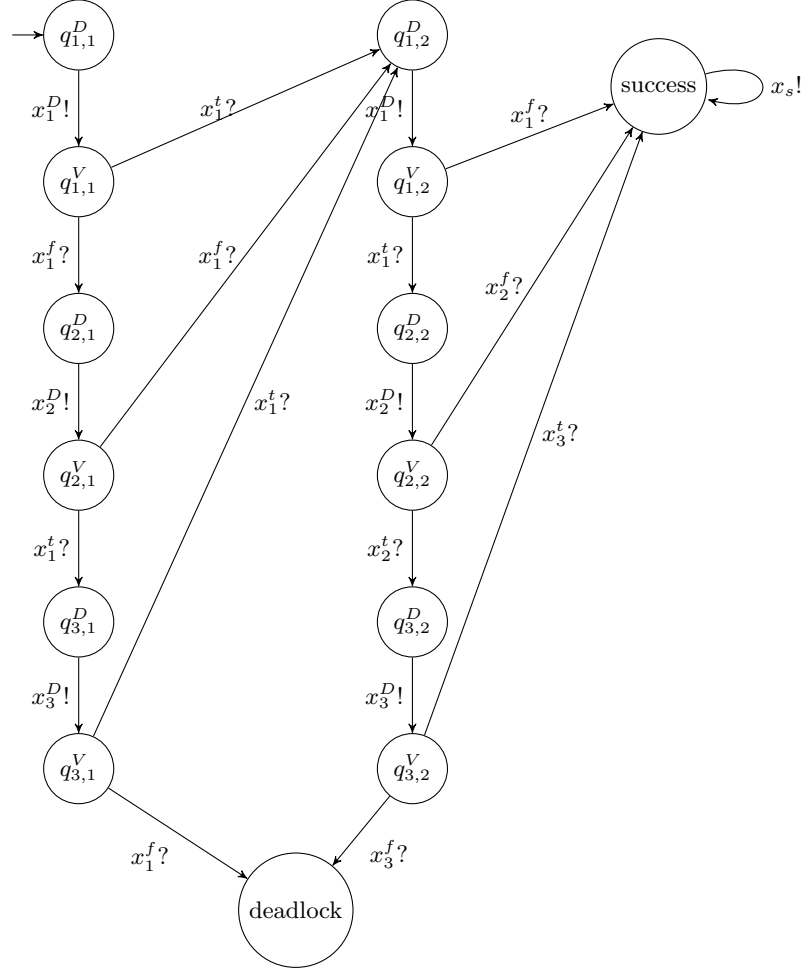


Fig. 9. Environment Automaton

each label x_i^D . Therefore P' is deterministic. We now show that $E \parallel P'$ has no reachable deadlock states.

Let R be the sequence of states in a run of $E \parallel P'$. If k is even then $R(k)$ is of the form $(q_{i,j}^D, q_0^P)$, $(success, q_0^P)$, or $(deadlock, q_0^P)$. In the first case, $R(k)$ has exactly one outgoing edge and $R(k+1)$ is of the form $(q_{i,j}^V, q)$, where if $u_k = v_j^i$, q is either q_k^t or q_k^f . In the second case $R(k+1) = R(k)$ and in the third case $R(k)$ has no outgoing transitions. By construction of E and P' , $(q_{i,j}^V, q)$, where q is either q_k^t or q_k^f , has exactly one outgoing transition. Therefore, if $(deadlock, q_0^P)$ is not reachable, then $E \parallel P'$ has no reachable deadlock states.

We assume that $(\text{deadlock}, q_0^P)$ is reachable and arrive at a contradiction. If it is then there is $j \in [1, m]$ such that the following hold: $(q_{1,j}^V, q_1) \xrightarrow{x_1^!} (q_{2,j}^D, q_0^P)$, $(q_{2,j}^V, q_2) \xrightarrow{x_2^!} (q_{3,j}^D, q_0^P)$, and $(q_{3,j}^V, q_3) \xrightarrow{x_3^!} (\text{deadlock}, q_0^P)$, where one of the following holds for each pair q_i and x_i :

- $z_j^i = u_k$ and $q_i = q_k^f$ and $x_i = x_k^f$,
- $z_j^i = \overline{u_k}$ and $q_i = q_k^t$ and $x_i = x_k^t$.

Furthermore, by the construction of T , $(q_{i,j}^V, q_k^f)$ is reachable if $(q_0^P, x_k^D, q_k^f) \in T$ or if $t(u_k) = \text{false}$, and $(q_{i,j}^V, q_k^t)$ is reachable if $(q_0^P, x_k^D, q_k^t) \in T$ or if $t(u_k) = \text{true}$.

We have shown that there is $j \in [1, m]$ such that for every $i \in [1, 3]$, if $z_j^i = u_k$ then $t(u_k) = \text{false}$ and if $z_j^i = \overline{u_k}$ then $t(u_k) = \text{true}$ which implies that $t(c_j) = \text{false}$ and contradicts t being a satisfying assignment.

Problem 1 \implies 3SAT: Assume T is a set of transitions, such that P' is the completion of P with T , P' is deterministic, and $E \parallel P'$ has no reachable deadlock states.

We construct a truth assignment t for U that satisfies all the clauses in C .

Since $E \parallel P'$ is deadlock free, for every $i \in [1, 3]$ and $j \in [1, m]$, if a state $(q_{i,j}^V, q)$ is reachable, because the only outgoing transitions from $q_{i,j}^V$ in E are input transitions with labels x_k^t and x_k^f , where $u_k = v_j^i$, and the only two states in P' that have outgoing output transitions with such labels are q_k^t and q_k^f , then q has to be q_k^t or q_k^f .

Similarly, the only outgoing transition from $q_{i,j}^D$ in E , if $u_k = v_j^i$, is an output transition with label x_k^D , and the only state in P' that can have an outgoing input transition with label x_k^D is q_0^P . Therefore, if $(q_{i,j}^D, q)$ is reachable in $E \parallel P'$ then q is q_0^P .

Combining the last two observations, and because $q_{i,j}^V$ always follows $q_{i,j}^D$ in E , if a state $(q_{i,j}^V, q)$ is reachable, then either $(q_0^P, x_k^D, q_k^t) \in T$ or $(q_0^P, x_k^D, q_k^f) \in T$.

We construct a truth assignment t for U as follows: for every u_k such that there exists i and j such that $v_j^i = u_k$ and $(q_{i,j}^V, q)$ is reachable, we set $t(u_k)$ to *true* if $(q_0^P, x_k^D, q_k^t) \in T$ and to *false* if $(q_0^P, x_k^D, q_k^f) \in T$. We assign an arbitrary value to the remaining variables in U . t is well-defined because P' is deterministic and thus it cannot be the case that both $(q_0^P, x_k^D, q_k^t) \in T$ and $(q_0^P, x_k^D, q_k^f) \in T$.

We show that t satisfies all clauses in C .

No deadlock states are reachable in $E \parallel P'$, therefore $(\text{deadlock}, q)$ is not reachable.

Therefore for each $j \in [1, m]$ there is $i \in [1, 3]$ such that one of the following holds:

- $z_j^i = u_k$ and $(q_{i,j}^V, q_k^t)$ is reachable or
- $z_j^i = \overline{u_k}$ and $(q_{i,j}^V, q_k^f)$ is reachable.

This guarantees that whenever $(q_{1,j}^D, q_0^P)$ is reachable in a run of $E \parallel P'$, the run continues to $(q_{1,j+1}^D, q_0^P)$ or $(success, q_0^P)$ and does not reach $(deadlock, q_0^P)$.

In the first case, it has to be that $(q_0^P, x_k^D, q_k^t) \in T$ or $t(u_k) = true$.

In the second case, it has to be that $(q_0^P, x_k^D, q_k^f) \in T$ or $t(u_k) = false$.

Hence, for every $j \in [1, m]$ there is $i \in [1, 3]$ such that if $z_j^i = u_k$ then $t(u_k) = true$ and if $z_j^i = \overline{u_k}$ then $t(u_k) = false$, or t satisfies all clauses in C .

Claim. Problem 2 is PSPACE-complete.

Proof. Problem 2 is in NPSPACE since we can guess a set of completions, and then check whether the requirements of the problem are satisfied. Checking whether the resulting process automata are deterministic can be done in polynomial time. Checking whether the product automaton is safe, empty, and deadlock-free can all be done in PSPACE. Since $NPSPACE = PSPACE$, Problem 2 is in PSPACE.

PSPACE-hardness can be seen by observing that a special case of Problem 2 is when the set of process automata is empty. This special case of determining the safety of a set of environment automata is at least as hard as reachability, which is PSPACE-hard.